Algorithms and Data Structures 2
CS 1501

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez’s and Dr. Farnan’s CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Midterm Question **Reattempts**: tonight @ 11:59 pm
  - **Lab 10**: ~~Tuesday 4/11~~ May 1 @ 11:59 pm
  - **Lab 11**: ~~Tuesday 4/18~~ May 1 @ 11:59 pm
  - **Lab 12**: May 1 @ 11:59 pm
  - **Homework 11**: ~~Friday 4/14~~ May 1 @ 11:59 pm
  - **Homework 12**: May 1 @ 11:59 pm
  - **Assignment 4**: ~~Friday 4/14~~ May 1 @ 11:59 pm
    - Support video and slides on Canvas + Solutions for Labs 8 and 9
  - **Assignment 5**: May 1 @ 11:59 pm
    - to be posted tonight

Final Exam

- **Friday 4/28 12:00-13:50**
  - 169 Crawford Hall
- Same format as midterm
- **Non-cumulative**
- Study guide and practice test on Canvas
- **Review Session** during Finals' Week
  - Date and time TBD
  - recorded

Bonus Opportunities

- **Bonus Lab**
  - worth up to 1%
  - lowest two labs still dropped
- **Bonus Homework**
  - worth up to 1%
  - lowest two homework assignments still dropped
- bonus point for class when

  OMETs response rate >= 80%
  - Currently at 12%
  - Deadline is Sunday 4/23

Previous Lecture

Dynamic Programming: Typical question in **coding interviews**!

- More Examples:
  - 0/1 Knapsack
  - Change Making
  - Subset Sum
  - Edit Distance

This Lecture

Dynamic Programming:

Typical question in **coding interviews**!

- More Examples:
  - Longest Common Subsequence
  - Reinforcement Learning
- **Maximum Flow** Problem: useful for problem solving
  - Ford Fulkerson
  - Push Relabel

Example 7: Longest Common Subsequence

- Given two sequences, return the **longest common subsequence**
  - `Example:`
    - `A Q S R J K V B I`
      `Q B W F J V I T U`
    - `A` **`Q`** `S R` **`J`** `K` **`V`** `B` **`I`**
      `Q` `B W F` **`J V I`** `T U`
- We'll consider a **relaxation** of the problem and only look for the ***length*** of the longest common subsequence

LCS dynamic programming example

`x = A Q S R J B I`

`y = Q B I J T U T`

i\j		Q	B	I	J	T	U	T
A								
Q								
S								
R								
J								
B								
I								

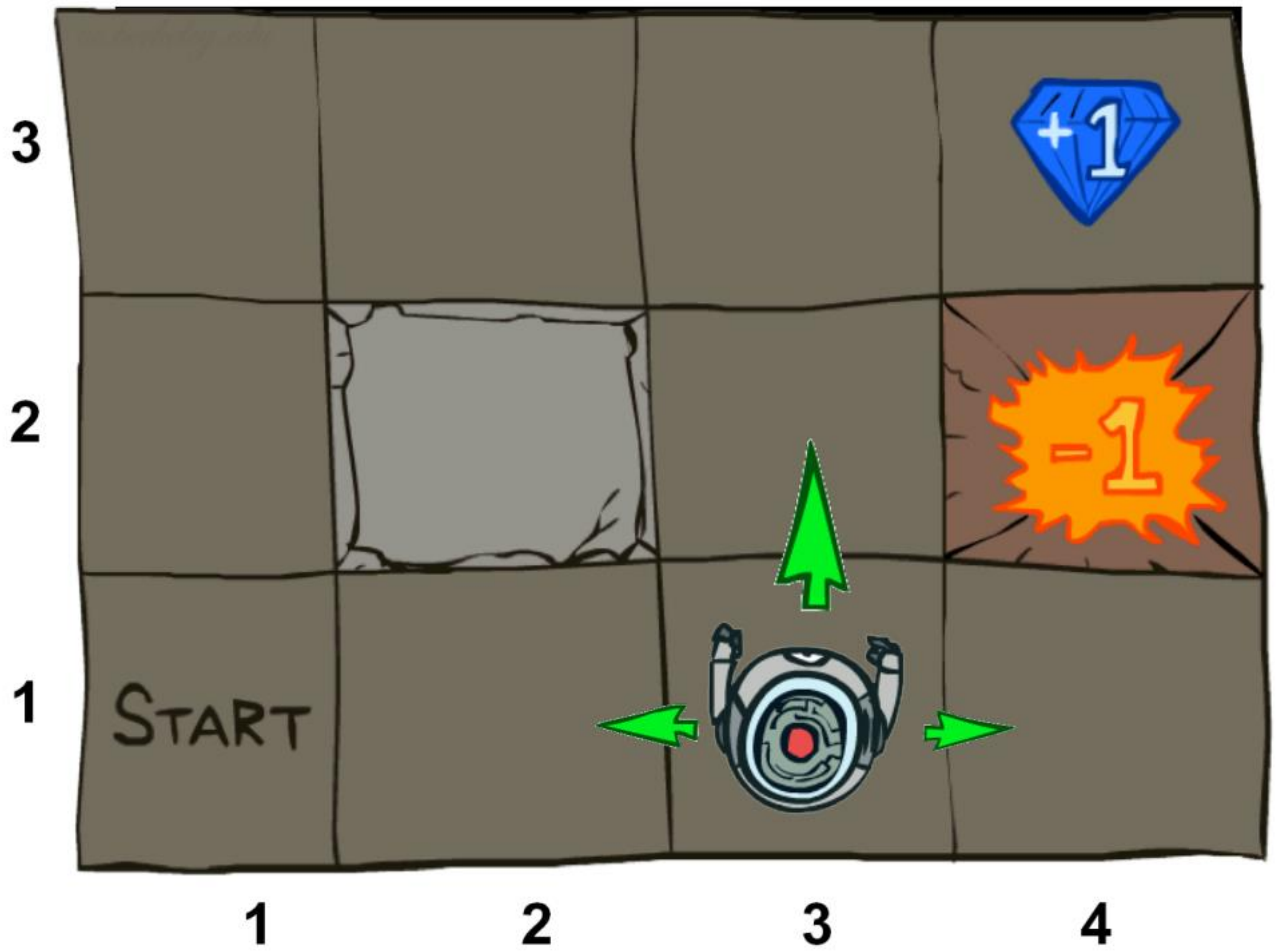
LCS dynamic programming solution

```
int LCSLength(String x, String y) {
   int[][] m = new int[x.length + 1][y.length + 1];
   for (int i=0; i <= x.length; i++) {
           for (int j=0; j <= y.length; j++) {
                 if (i == 0 || j == 0) m[i][j] = 0;
                 if (x.charAt(i) == y.charAt(j))
                       m[i][j] = m[i-1][j-1] + 1;
                 else
                       m[i][j] = max(m[i][j-1], m[i-1][j]);
           }
     }
   return m[x.length][y.length];
}
```

Example 8: Reinforcement Learning

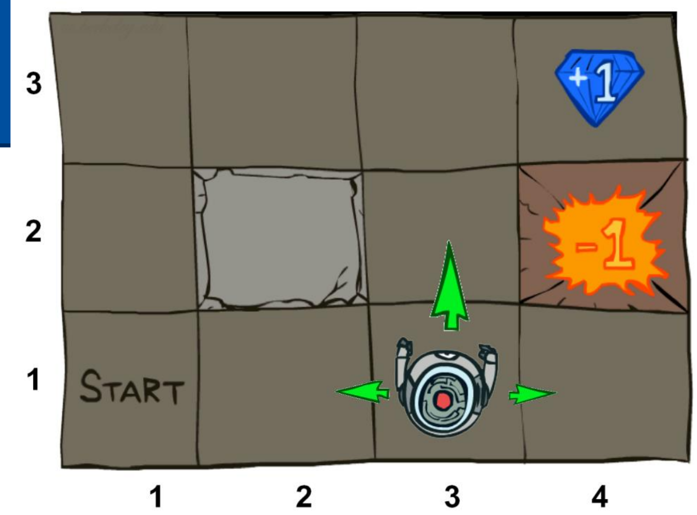
- A type of **Machine Learning**
  - an **agent** (e.g., a robot)
  - learns an optimal **policy**
  - only by getting **rewards** from the **environment**

Example



Source: https://inst.eecs.berkeley.edu/~cs188/sp20/assets/lecture/lec10.pdf

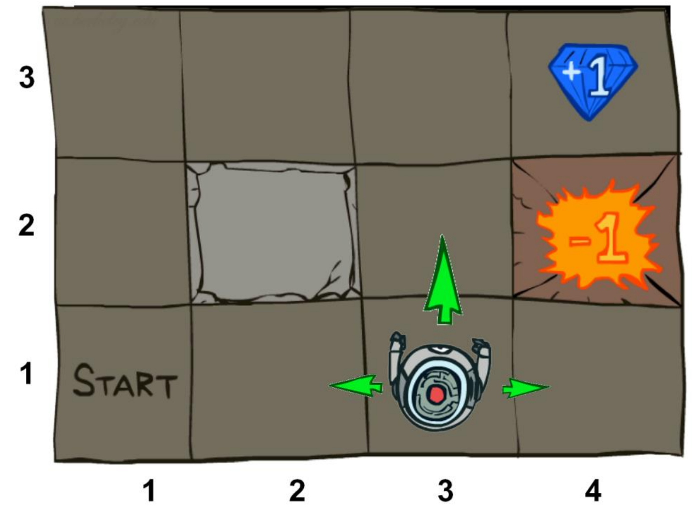
Input: Markov Decision Process



- A set of **states**
  - e.g., maze locations, agent health
- A set of agent **actions**
  - e.g., move left, move right, etc.
- **Probabilities** of ending up in a state given a current state and an action
  - e.g., move left action → moving left with 1.0 prob. if no wall
  - e.g., if wall, move left action → moving right or up or down with 0.33 prob.
- **Reward** function
  - depends on state and action
  - e.g., high reward for moving up from below cheese
- **Starting state**

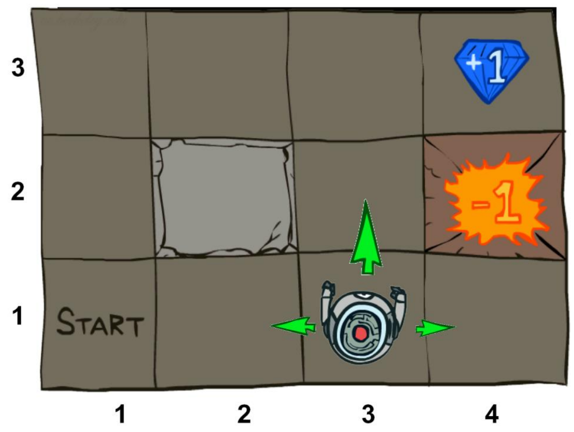
Input: Markov Decision Process

- A set of **states**
  - think graph **vertices**
- A set of agent **actions**
  - think graph **edges**
- **Probabilities** of ending up in a state given a current state and an action
- **Reward** function
  - think edge **weights**
- A special case: all information readily available
  - called **Planning**

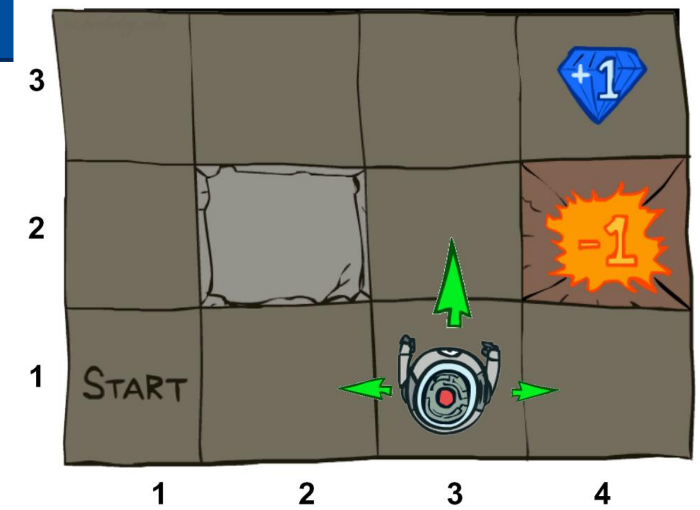


Output: Optimal Agent Policy

- An agent policy determines the **probability** of taking an action given a state
  - e.g., prob. 1.0 for moving left from start
- An optimal policy gives the **maximum total reward**
- Let’s embed rewards into **state values**
  - think distance[] in **Bellman**-Ford
- An optimal policy gives the **maximum total state value**



Expectations



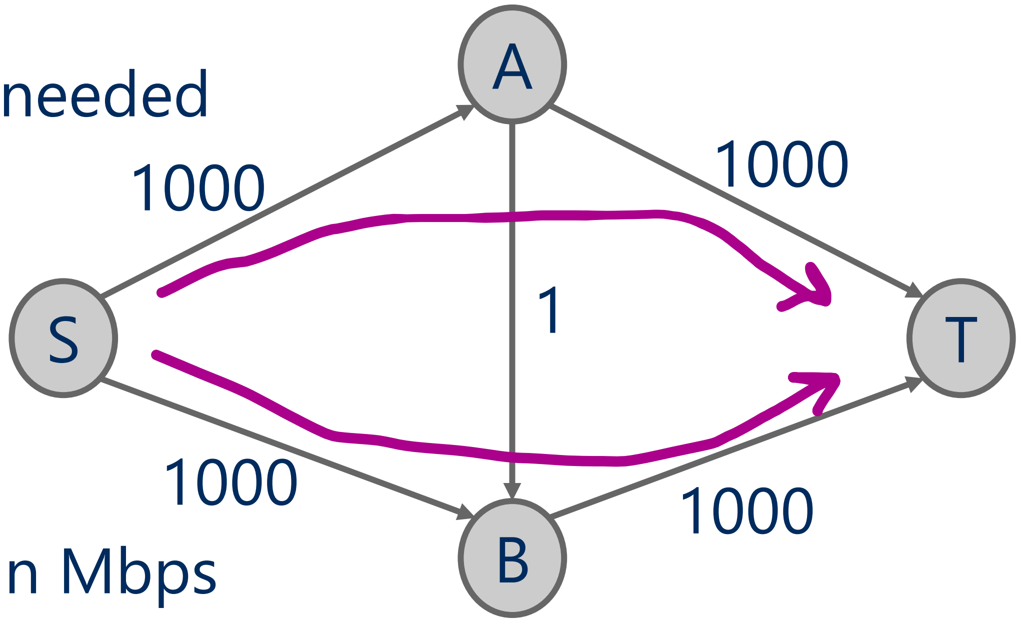
- Expected **value** of a state?
  - depends on actions
  - Max_{all actions}:
    - prob. of action **(from policy)** * expected reward
- Expected reward from an action depends on
  - immediate reward (from reward function)
  - values of states reachable through the action
  - Sum_{all states}:
    - prob. of reaching state * **state value**

Using Dynamic Programming to Solve an MDP

- Data Structure: Array of state values
- Step 0: Start with an **initial** policy and initial state values
  - e.g., all actions equally likely and state values = 0
- Step 1: Compute expected state values
  - optional: **iterate** until values **converge**
- Step 2: **Modify** policy to take the best action with probability 1.0 (given the current state values)
- Repeat Step 1 and 2 until policy **converges**

Problem of the Day: Finding Bottlenecks

- send a large file from S to T over a computer network
  - as fast as possible
  - over multiple network paths if needed
- Input:
  - computer network
    - nodes and links
    - links labeled by link speed in Mbps
  - nodes A and B
- Output:
  - The **maximum network speed** possible



Defining flow network

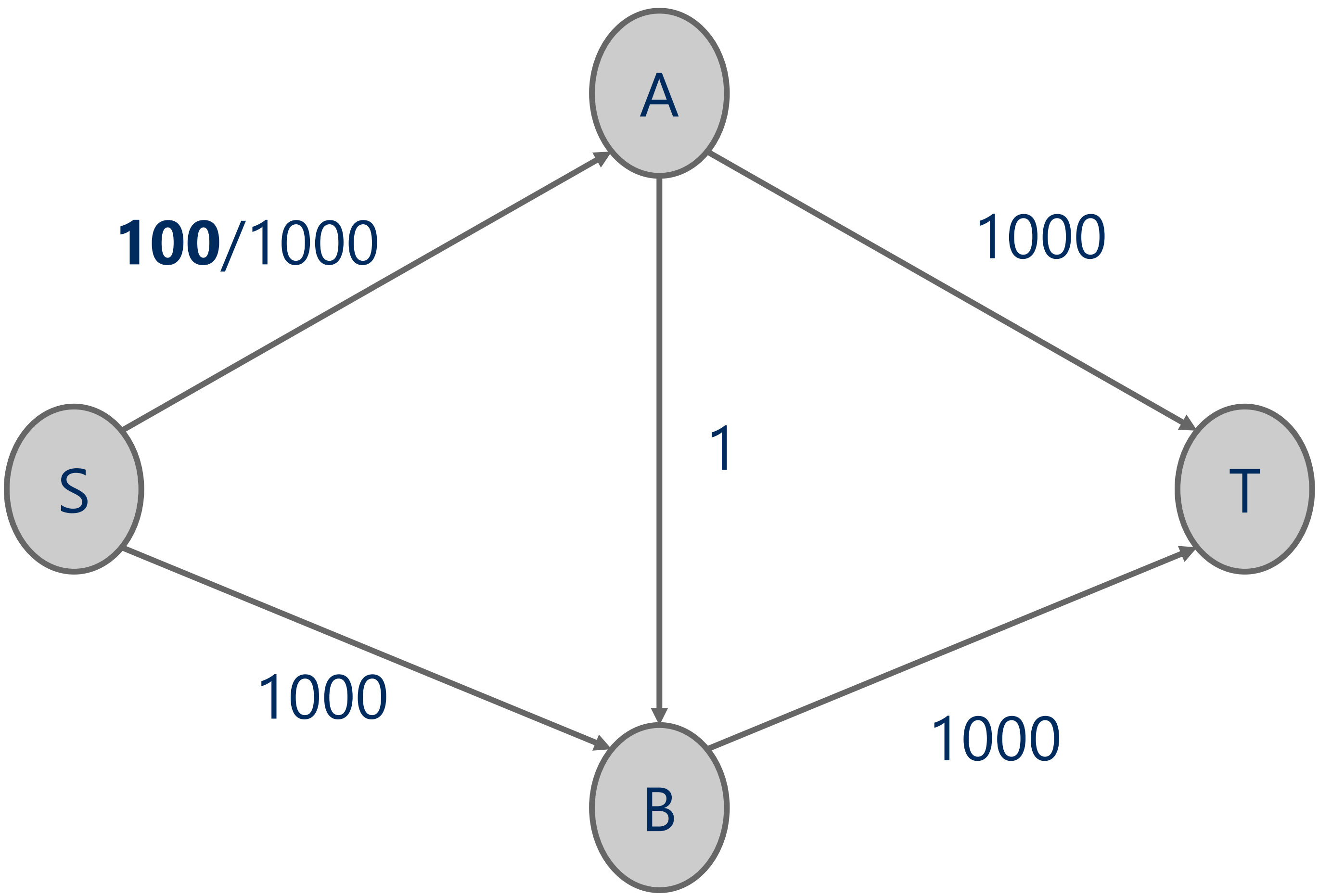
- directed, weighted graph G(V, E)
    - Weights are applied to edges to state their *capacity*
        - c(u, w) is the capacity of edge (u, w)
        - if there is no edge from u to w, c(u, w) = 0
- Consider two vertices, a *source s* and a *sink t*
    - determine maximum flow from s to t in G
    - with constraints!

Flow

- f(u, w): amount of flow carried along the edge (u, w)
- Some rules on the flow running through an edge:
  - **$f(u, w) <= c(u, w)$**
    - $\forall (u, w) \in E$
  - **$\Sigma_{w \in V} f(w, u) = \Sigma_{w \in V} f(u, w)$**
    - incoming flow = outgoing flow
    - $\forall u \in (V - \{s,t\})$

Residual Capacity of an edge

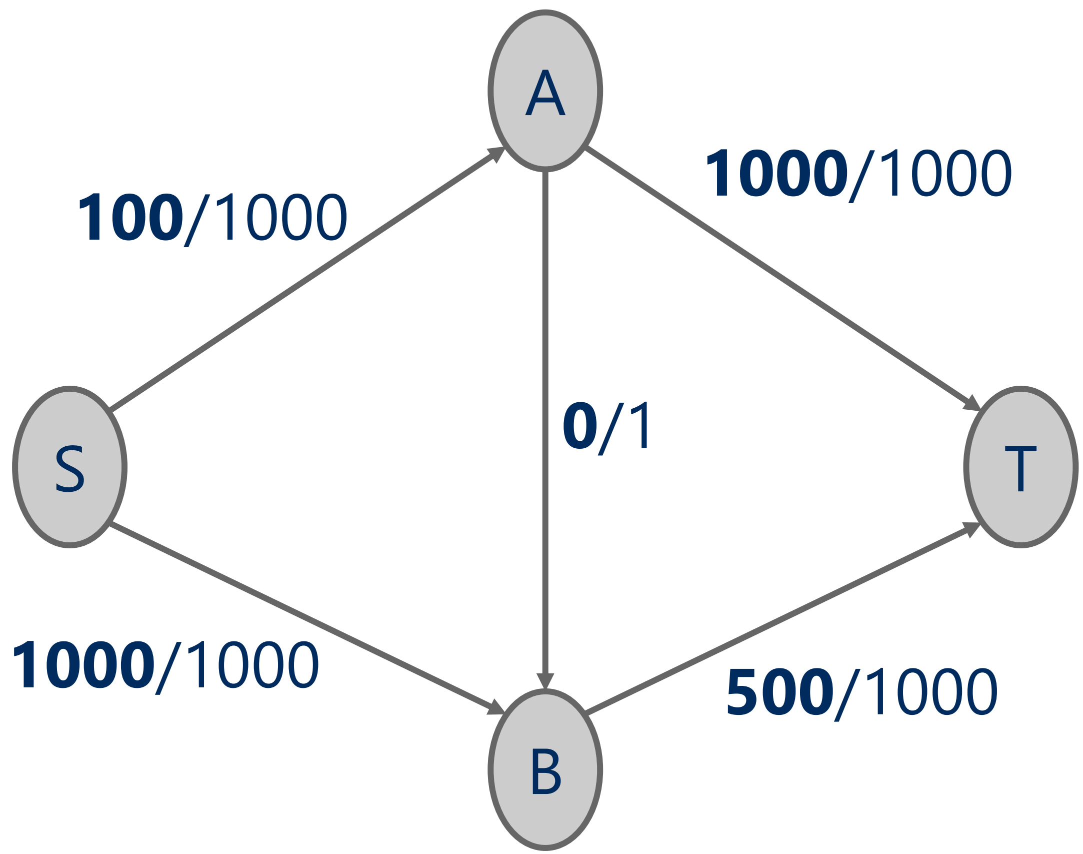
- Residual capacity of edge (u, w) is **c(u, w) - f(u, w)**
- e.g., residual capacity of edge (S, A) is 1000 – 100 = 900



Augmenting Path

An **augmenting path** in G:

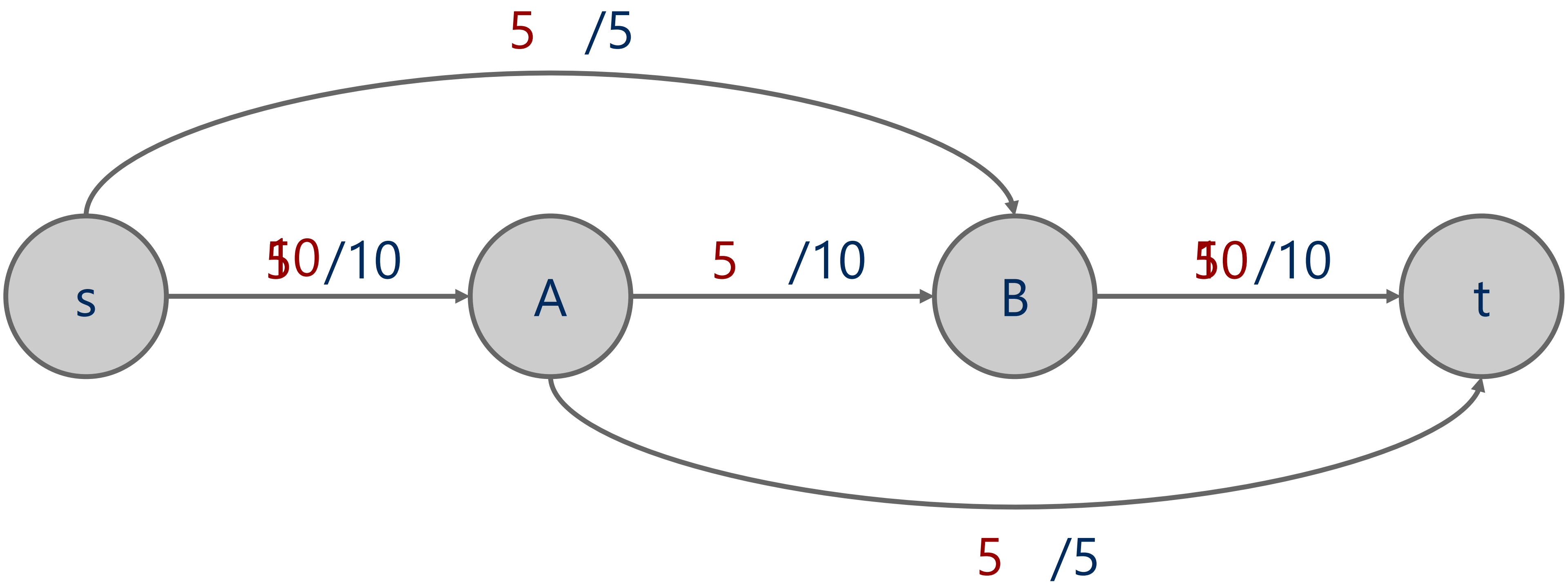
- simple path *p* from s to t
- all edges in p have some **residual capacity**
  - $\forall(u, w) \in p$, $\mathbf{f(u, w) < c(u, w)}$
- **S → A → B → T**
- **S → A → T**
- **S → B → T**



Ford Fulkerson

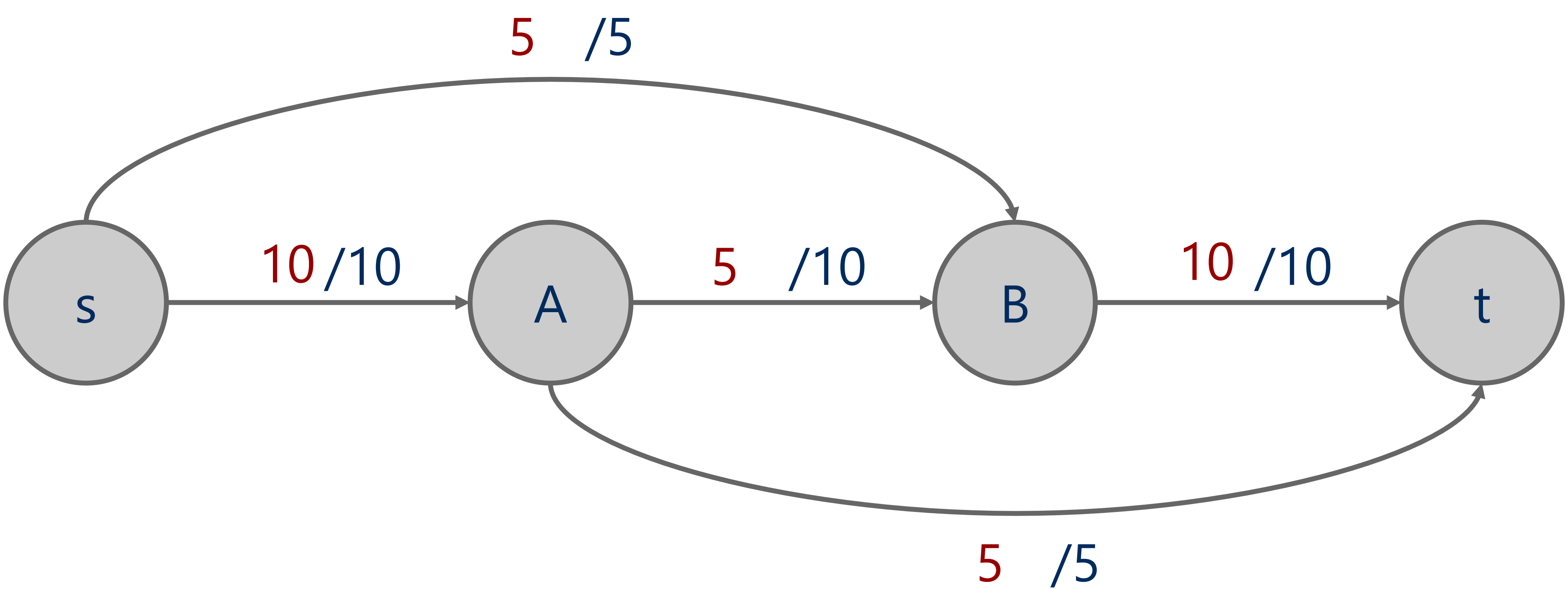
- Initially: $\mathbf{f(u, w) = 0}$ $\forall(u, w) \in E$
- While an augmenting path p exists
  - Find an edge with **minimum residual capacity** in p
    - call this minimum residual capacity *new_flow*
  - **Increase** the flow on all edges in p by *new_flow*

Ford Fulkerson example

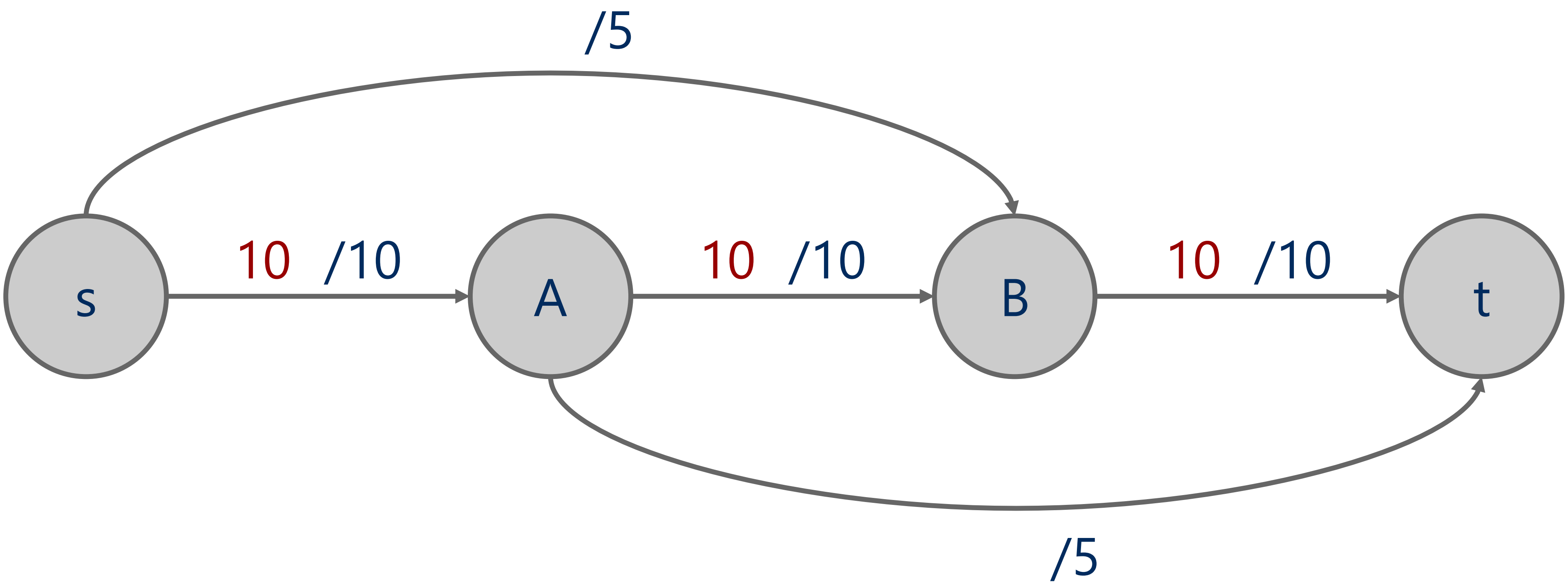


So what is the value of the maximum flow?

- Add up the **flow increments** in iterations of Ford-Fulkerson
- Add up the edge **<u>flows</u> out of source**
- Add up the edge **<u>flows</u>** of **into sink**



What if we selected augmenting paths in different order?



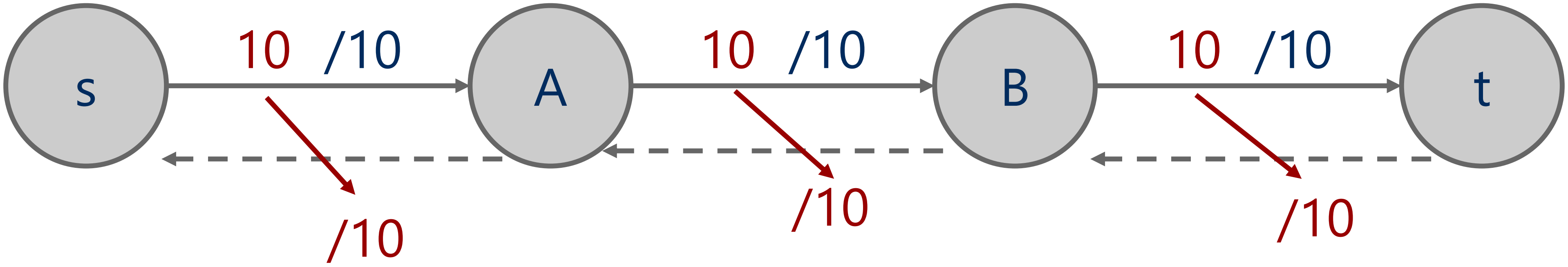
Can’t reach maximum flow value of 15!

How does Ford-Fulkerson correct that?

- consider **re-routing** previously allocated flow
- when finding an augmenting path, look not only at the edges of G, but also at **backwards edges** that allow such re-routing

Backwards Edges

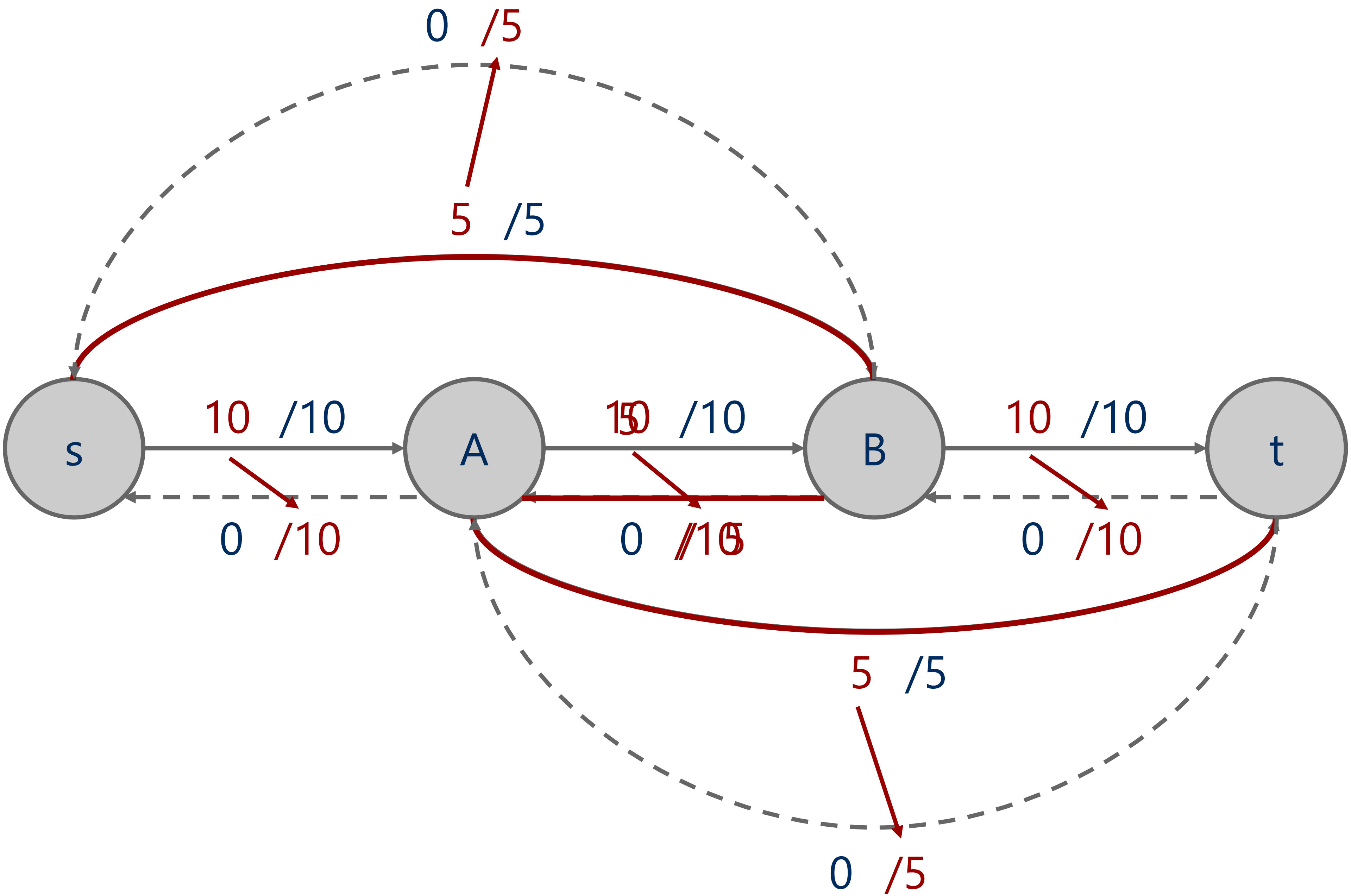
- For each edge (u, w) ∈ E with **f(u, w) > 0**, a backwards edge (w, u) exists
- The capacity of the backwards edge (w, u) **= f(u, w)**
- Adding flow to a backwards edge means **rerouting** flow from the corresponding forward edge



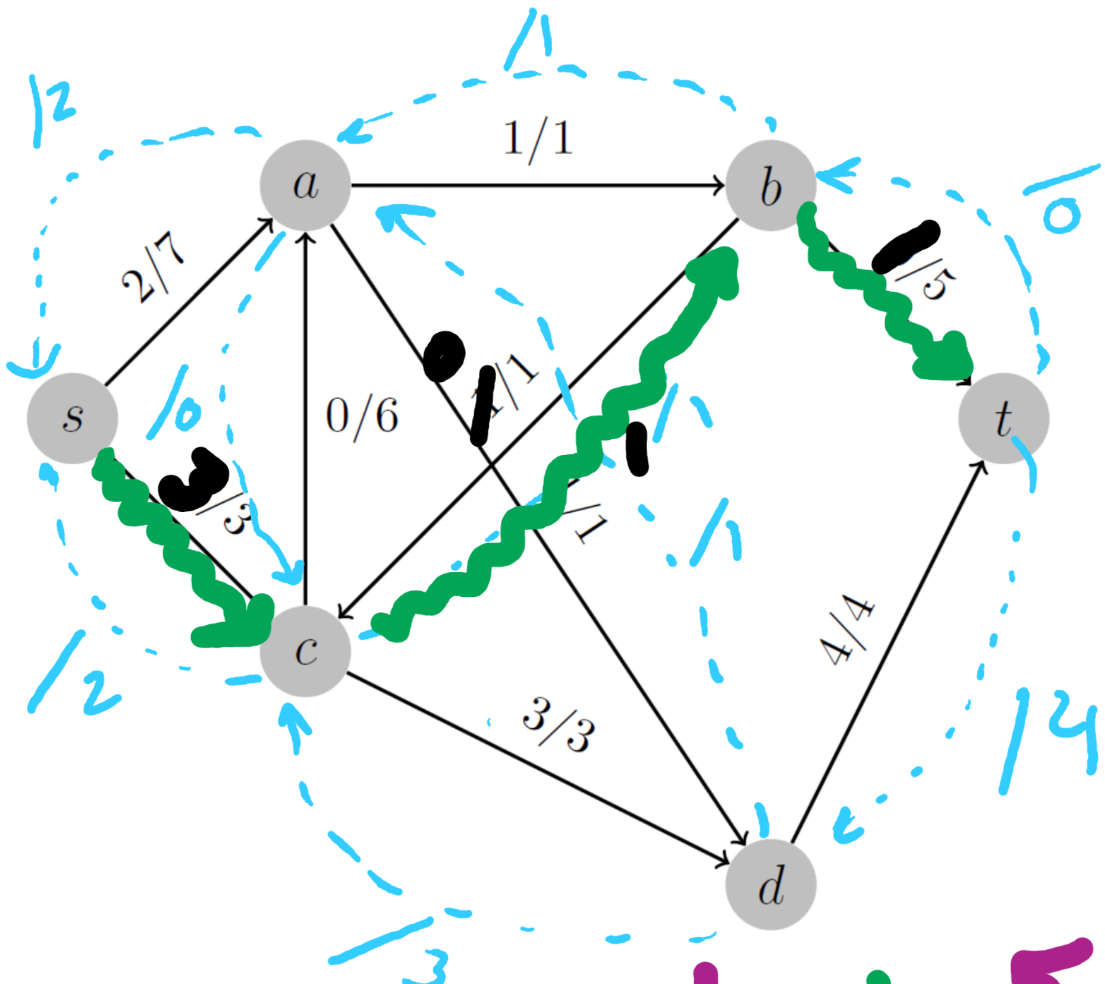
The residual graph

- searches for augmenting path on a **residual graph**
- The residual graph is made up of:
    - V
    - An edge for each (u, w) ∈ E where **f(u, w) < c(u, w)**
        - 0 flow and a capacity of c(u, w) - f(u, w)
    - A **backwards edge** for each (u, w) ∈ E where **f(u, w) > 0**
        - (u, w)'s backwards edge has a capacity of f(u, w)
        - All backwards edges have 0 flow

Residual graph example

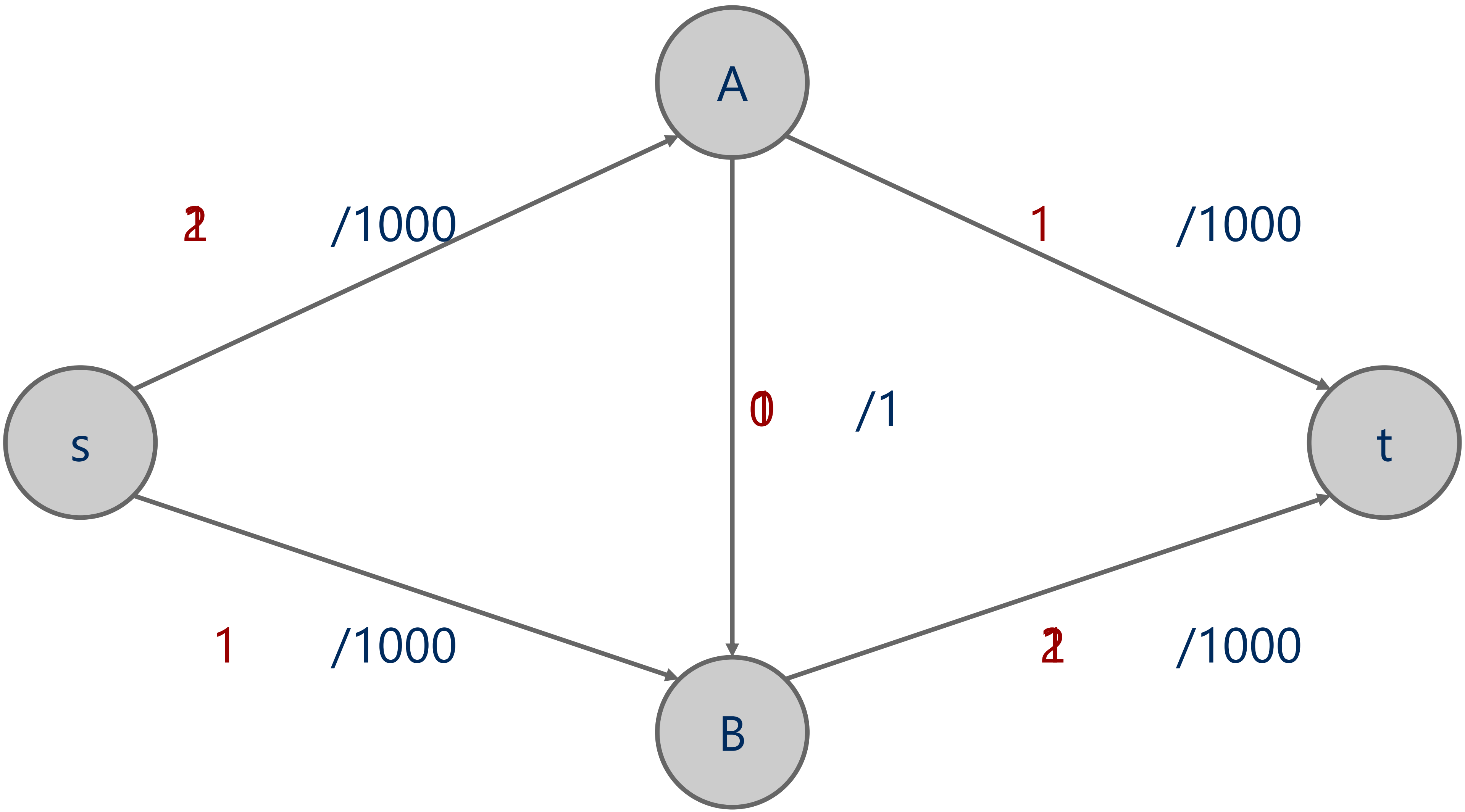


2nd Tophat Question



$s \xrightarrow{1} c \xrightarrow{1} b \xrightarrow{5} t$

Another example



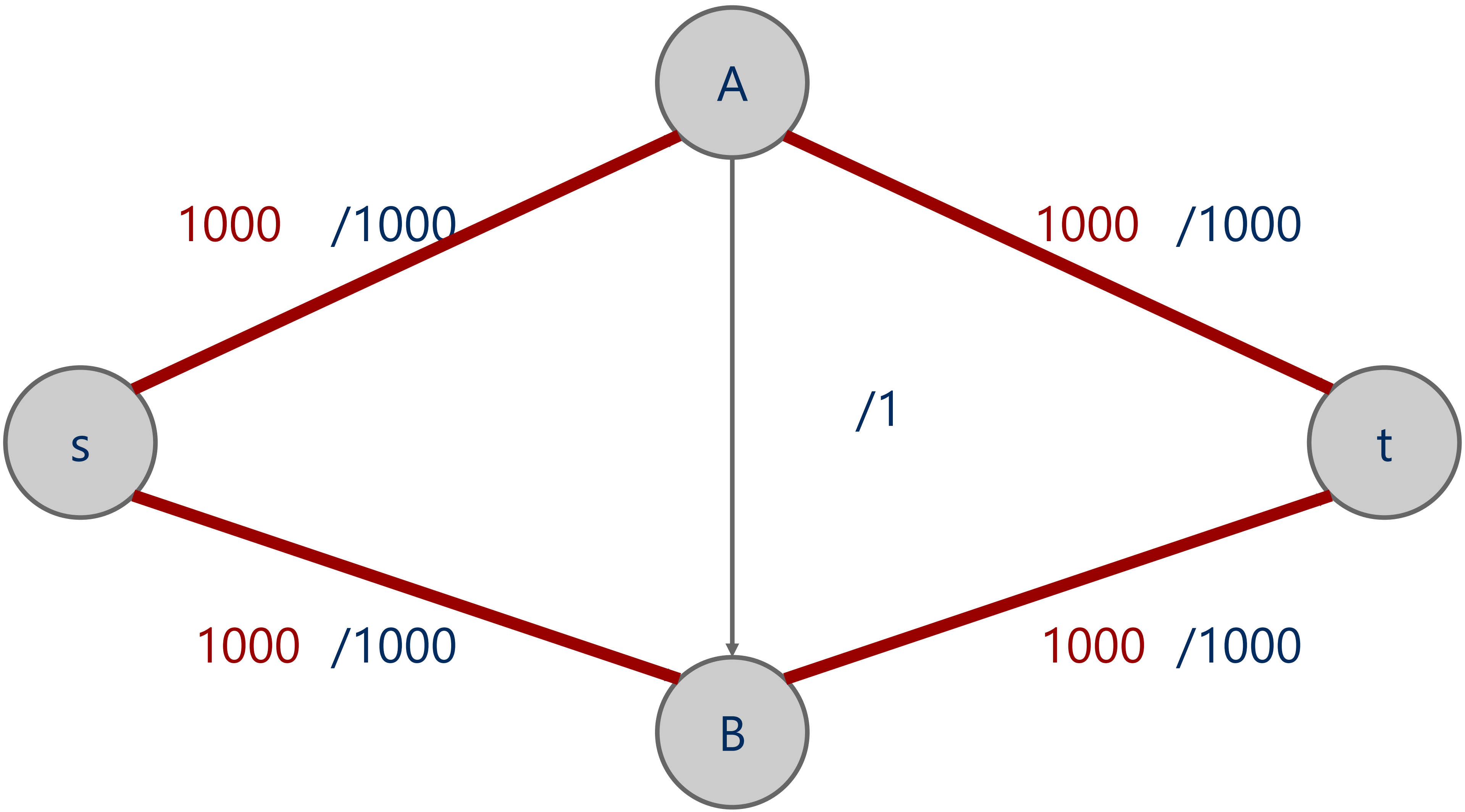
Worst-case Runtime of Ford-Fulkerson

- $O(f * (e+v))$
- f: value of maximum flow
- e+v: time to find an augmenting path
- is that polynomial in the input size?
- No! f is exponential in bitlength of f
- $O(2^{|f|} * (e+v))$

Edmonds Karp

- How the augmenting path is chosen affects the **performance** of the search for max flow
- Edmonds and Karp proposed a **shortest path heuristic** for Ford Fulkerson
  - Use **BFS** to find augmenting paths

Another example



Edmonds Karp

- Running time is $O(e^2 v)$

But our flow graph is weighted...

Edmonds-Karp only uses BFS

- BFS finds spanning trees and shortest paths for **unweighted** graphs
- some **weight-based** measure of priority to find augmenting paths?

Maximum Capacity Path

- Proposed by Edmonds and Karp
- implemented by modifying Dijkstra's shortest paths algorithm
- Define **flow[v]** as the maximum amount of flow from s → v along a **single path**
- Each iteration, set curr as an unmarked vertex with the largest flow
- For each neighbor w of curr:
    - if **min(flow[curr], residual capacity of edge (curr, w))** > flow[w]
        - update flow[w] and parent[w] to be curr

Flow edge implementation

- For each **edge**, we need to store:
  - **from** vertex
  - **to** vertex
  - **edge capacity**
  - **edge flow**
  - residual capacities
    - For **forwards** and **backwards** edges

FlowEdge.java

```
public class FlowEdge {
   private final int v;                  // from
   private final int w;                  // to
   private final double capacity;    // capacity
   private double flow;                  // flow
     …
     public double residualCapacityTo(int vertex) {
      if       (vertex == v) return flow;
      else if (vertex == w) return capacity - flow;
      else throw new
       IllegalArgumentException("Illegal endpoint");
   }
   …
}
```

BFS search for an augmenting path (pseudocode)

```
edgeTo = [v]
marked = [v]
Queue q
q.enqueue(s)
marked[s] = true
while !q.isEmpty():
    v = q.dequeue()
    for each edge in AdjList[v]:
        if residualCapacity(other end-point) > 0:
            if !marked[w]:
                edgeTo[w] = v;
                marked[w] = true;
                q.enqueue(w);
```

Each FlowEdge object is stored in the adjacency list twice:

Once for its forward edge
Once for its backward edge

Push-Relabel Algorithm for Max Flow

- More **efficient** than Edmonds-Karp that uses BFS
  - Running time: **Theta(v^3)** vs. Theta(e^2v)
- **Local** per vertex operations instead of global updates
- Each vertex has a **height** and **excess flow** value

Push-Relabel Algorithm for Max Flow

- **push operation:**
  - Flow **pushed** from higher vertex to lower neighbor
    - height difference of 1
    - over an edge with **residual capacity > 0**
- **relabel operation:**
  - If a vertex's excess flow > 0 and has no lower neighbor
    - **relabel** vertex's height to
      - **1 + min height of neighbors able to receive flow**

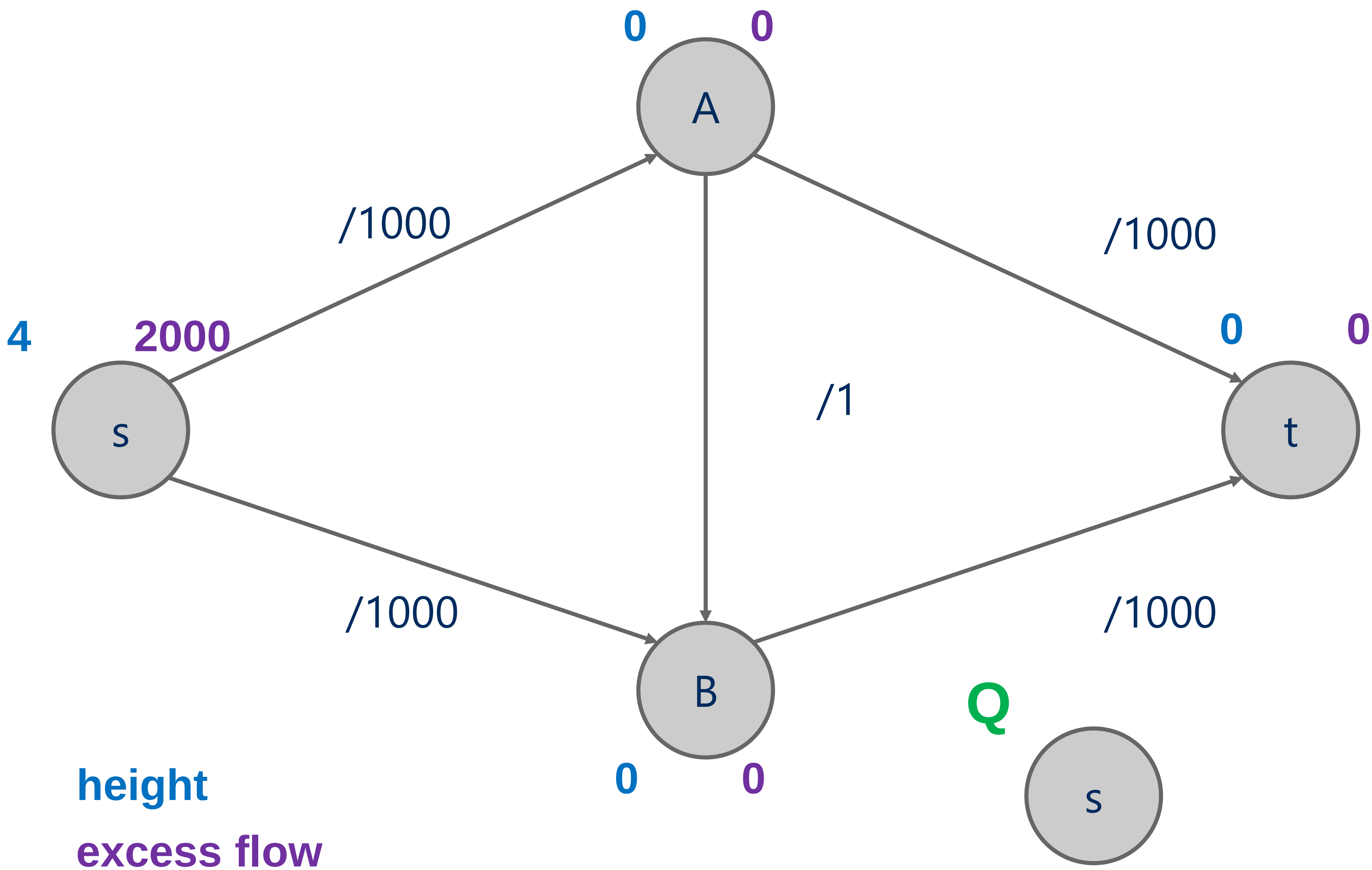
Push-Relabel Algorithm for Max Flow

- **push operation:**
- **relabel operation:**
- **Repeat** relabel and push operations until
  - all vertices except source and sink have **0 excess flow**

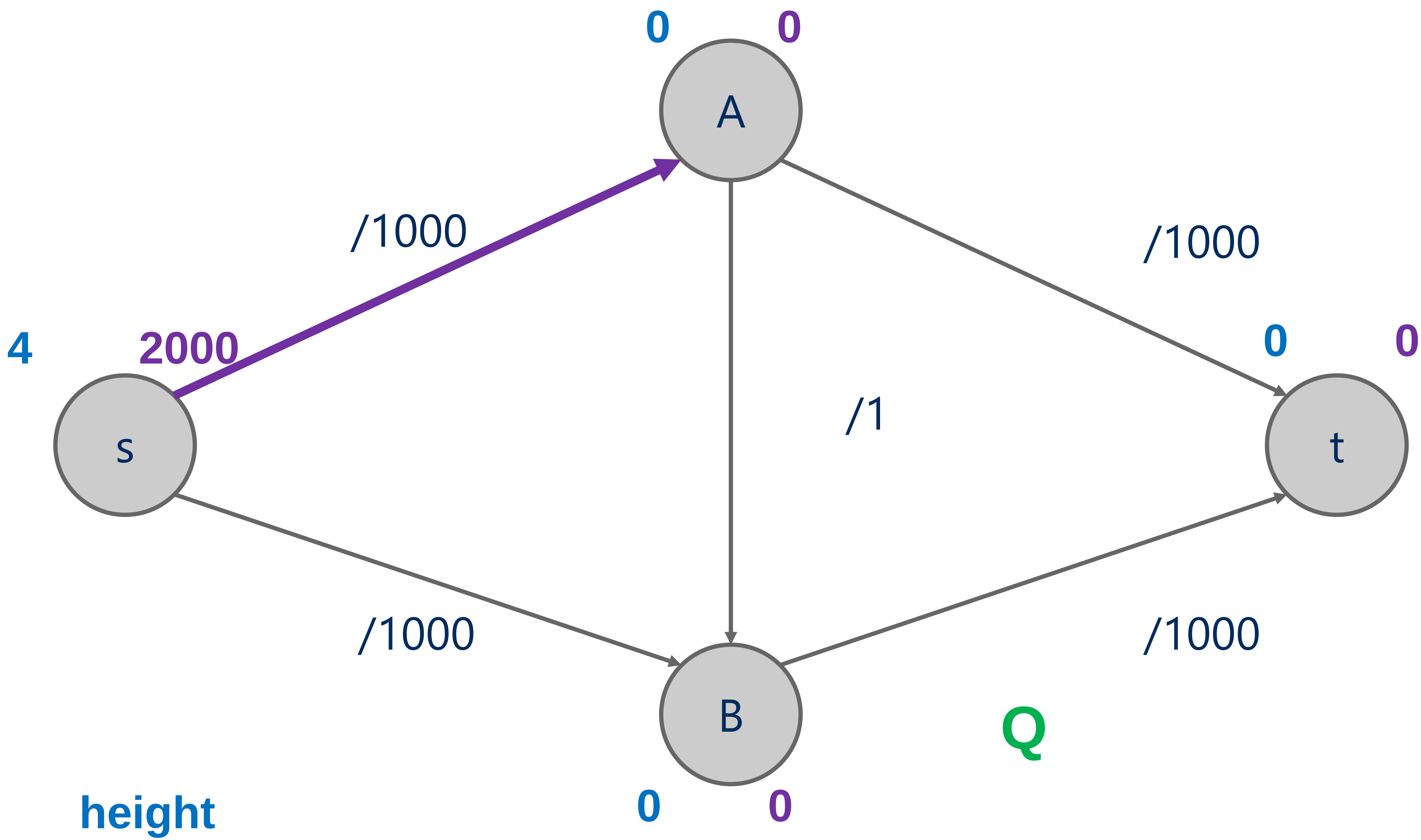
Push-Relabel Algorithm

- height = 0 and excess = 0 for all vertices
- excess[s] = sum of edge capacities out of s
- height[s] = v
- insert s into Q
- **while Q not empty**
  - v = pop head of queue
  - relabel v if needed
  - **for each neighbor w of v:**
    - push as much of v's excess flow to w
    - increase w's excess flow by the pushed amount
    - add w to Q if not already there

Push Relabel Example



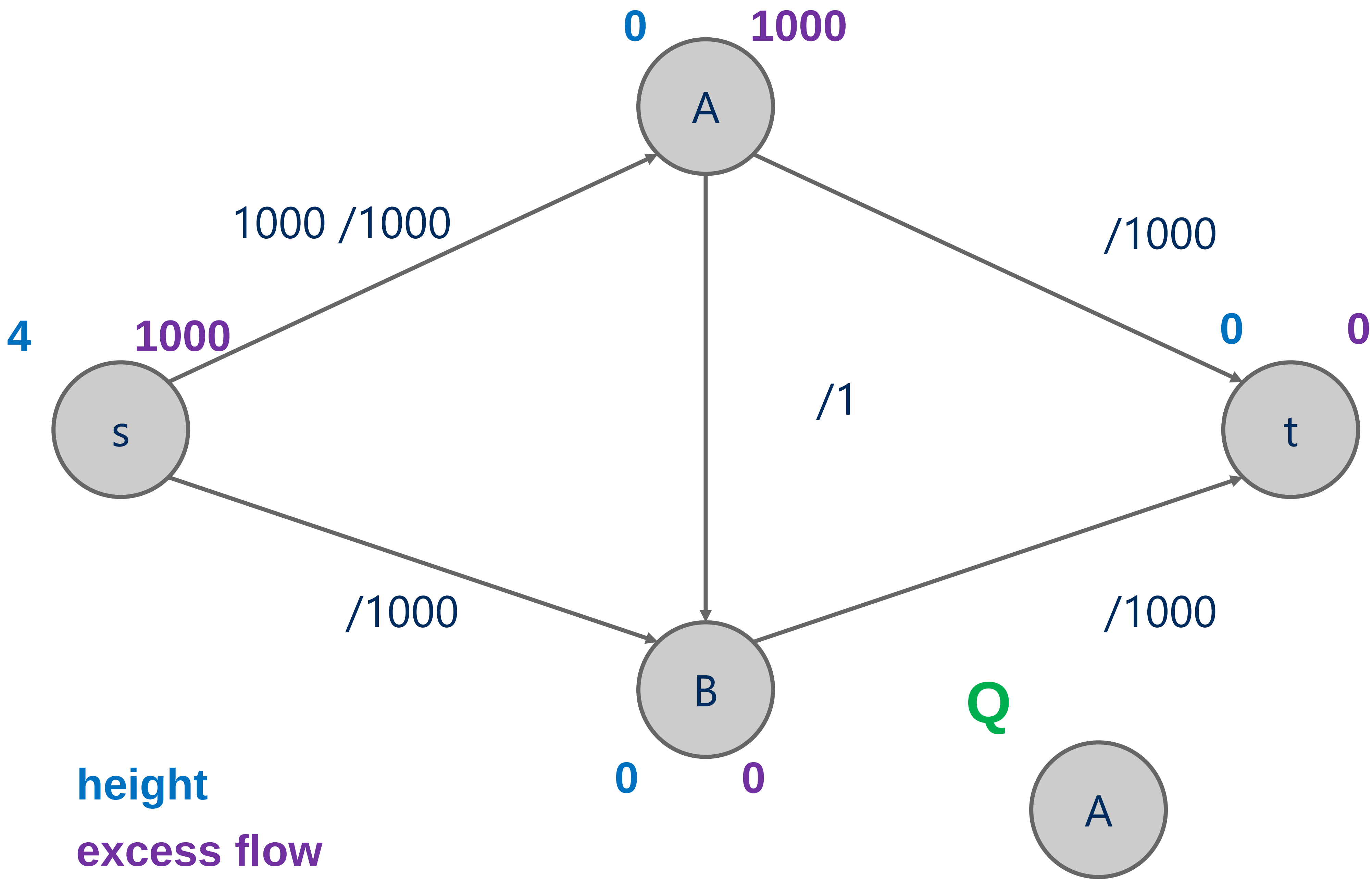
Push Relabel Example



height

excess flow

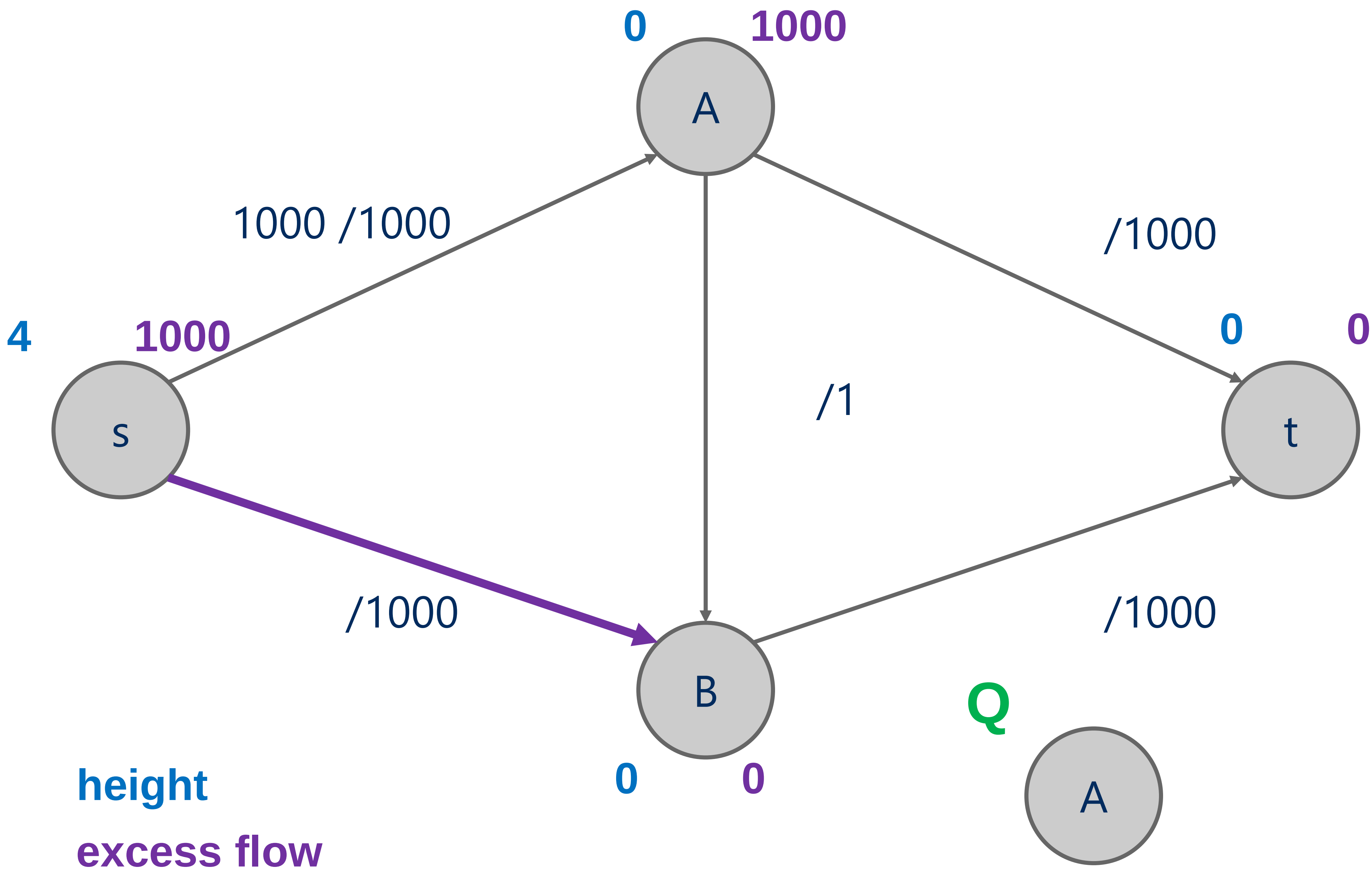
Push Relabel Example



height

excess flow

Push Relabel Example



Push Relabel Example